Homework3 Engineering and Error Analysis with UIMA

Jing Li (jli3)

Task 1. Baseline System Design

1.1 Overview

Generally, the system firstly reads the retrieval document and generates annotations for each document. Then, the system computes the term frequency for each candidate document. With all these information and data, the similarity between each query and its corresponding retrieval answer candidates are computed. Then, based on the similarity scores, the relevant document for each query is extracted from all the possible candidates through its settled relevance, and its rank is computed based on the comparison with the other candidates. At last, the Mean Reciprocal Rank (MMR) value can be computed and displayed as in the output file (report.txt).

1.2 Baseline System Componnets

The system baseline for task 1 is to compute cosine similarity between term frequency vectors, without any optimization by tokenizing method, punctuation and/or stop words removal, or any NLP techniques. Thus, the purpose this task 1 is to verify the correctness of the system implementation.

Key Components:

1.2.1 DocumentReader

The DocumentReader provided is able to extract the text of the whole document set and generate the document annotation for each document, including the relevance value, query id and the text content. It will finally output the document annotation into JCas pipeline for the system.

1.2.2 DocumentVectorAnnotator

The DocumentVectorAnnorator firstly generates the whole dictionary for the whole retrieval process. It will create term frequency vector for each document, which includes all the tokens and their corresponding term frequency. Finally, it will update all the document annotations' token list feature by adding term frequency value.

1.2.3 RetrievalEvaluator

The RetrievalEvaluator is responsible for calculating the similarity score of two doc term frequency vectors using provided similarity calculation methods (cosine similarity in the baseline case). Based on the similarity scores, the relevant document for each query is extracted from all the possible candidates through its settled relevance, and its rank is computed based on the comparison with the other candidates. At last, the Mean Reciprocal Rank (MMR) value can be computed and displayed as in the output file (report.txt). In conclusion, a data flow diagram can illustrate the architecture of the baseline system, as shown below in Figure 1.

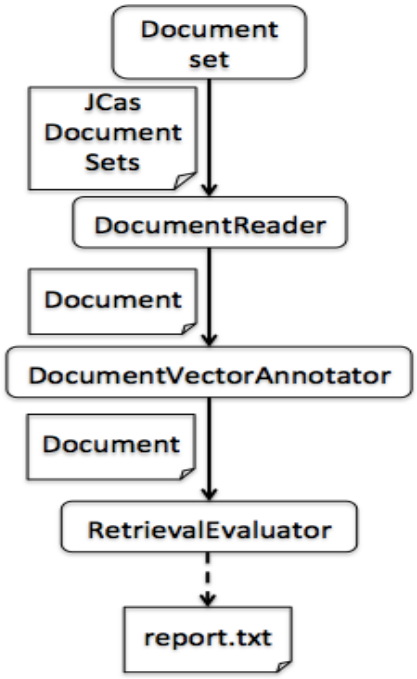

Figure 1. Baseline System Data Flow Diagram

1.3 Baseline System Performance (output sentence is omitted here)

From the output, we can retrieve the performance for the baseline system, in which MRR = 0.4375 in this case.

1.4 Summary

From the result shown as above, we can see that the performance of the baseline system is not so great, the error analysis and possible system improvement are described in next section.

Task 2. Error Analysis and System Improvement

2.1 Error Analysis

With observation of the input data, the errors that cause the low performance of the baseline system are analyzed and classified as below, and the corresponding examples are also given.

2.1.1 Token mismatch

The tokenizing method in baseline system is not working so great, which just generate the tokens based on the whitespace division. However, sometimes, this type of tokenizing leads to mismatch. For example, in query 2, the token “Jordan—one” in the relevant document cannot match to the token “Jordan” in the query, which leads to an error. Another example is that: the token “China's” in the query 5 cannot match the token “China” in its corresponding relevant candidate document. So tokening method is required to be improved and stemming should also be considered.

2.1.2 Tense difference

The different tense of a same verb can mean the same thing, which cannot be recognized by the baseline system and thus can be a source of error. For example, the token “bite” in the query 8 cannot match the token “bit” in its corresponding candidate document due to the difference tense. Also, the token “die” in the query 15 cannot match the token “died” in its corresponding candidate document due to the same reason. Thus, lemmatizer/stemming is definitely a great candidate method to improve the system performance.

2.1.3 Case-sensitive error

In the dataset, I found that for several documents, the reason that the match could not be found is that the upper/lower case differs between the query and the candidate for specific token. For example, the token “The” in the query 19 cannot be matched with the token “the” in corresponding candidate document. Also, the token “moon” in the query 9 and the token “Moon” in corresponding correct candidate document can also be another example. Thus, transfer all the cases to the lower cases might be helpful to improve the performance.

2.1.4 Punctuation

For some cases, the punctuation matters a lot. For example, in the query 11, the token "Devils" in the query cannot match "Devil's" in its corresponding candidate document, which also might be a spelling error. Another example is that in the query 6, the token "four minutes" in the query cannot match the "four-minute" in the correct relevant candidate document. Thus, punctuation removal and/or use punctuation as the second tokenizer except whitespace might be helpful to improve the system performance.

2.1.5 Synonyms

Synonyms can be another source of error, which is a quite hard type to get it right. For example, the token "spaceship" in the query 9 can be matched with its synonym "spacecraft" in its corresponding relevant candidate document, leading to an error. Also the token "deep" in the query 13 cannot match with the token "depth" in the correct candidate, which is also a type of synonyms.

2.2 System Improvement

Based on the observation and input data and the error analysis given as above, several possible optimization methods can be utilized to see if they can help to make the performance better.

2.2.1 Better tokenizing method, change to lower case and punctuation removal

First, since the performance of *tokenize0()* function required in the baseline system is not working so great, PTBTokenizer from stanford.nlp is used to see different tokenizing method can help improve the perfermance (simultaneously, all words extracted are transferred to lower case and the punctuation is removed as well).

After utilizing PTBTokenizer, changing all tokens to lower case and removing punctuation, the performance is highly improved (MRR = 0.7012). Thus, the methods applied here is very helpful to improve the performance.

Besides, this PTBTokenizer has a better performance than the Lucene package (all the processing included, stemming, stop words, lucene takenizer, etc.), which has a total

performance MRR = 0.654. It's possible that the lower performance is caused by the combined stop words removal in Lucene.

2.2.2 Stop words removal

Since the input document includes many meaningless words, a stop word removal might be helpful to improve the performance. Multiple ways can be utilized to delete stop words. I can either choose to delete them with Stanford NLP toolkit with their tokenizers, or I can just use the stop word dictionary provided in the archetype. I chose the latter. However, after applying the stop word removal, the performance drops (MRR = 0.6644). After looking into the file, I found that before the stop word removal, for several specific queries, the correct candidate wins just because that it has a shorter sentence, while after the removal of the stop words, a false candidate wins because specific stop words appear more than once, which can significantly affect the calculation of the similarity score. Thus, this method is not helpful for this specific given dataset.

2.2.3 Stemming/lemmatizer

The simple vector model does not understand that the different tenses or forms of a same word denote the same thing. Thus, the Stanford lemmatizer provided is applied to see if stemming can help improve the system performance. After applying this lemmatizer (without the removal of stop words based on the observation that stop word removal does not help), the performance did improve, but the effect is not enormous (MRR = 0.7042).

Task 3. Bonus: different similarity measures

3.1 Introduction

Here, except the Cosine Similarity required, I choose to implement two different similarity calculation methods to measure the similarity between the term vectors, Dice Similarity and Jaccard Similarity with all the improvement parts appended. In order to incorporate these two components efficiently and reasonably, a basic interface computing similarity is designed and implemented. In the system, different similarity tools can inherit the same similarity interface and just override the computeSimilarity() function with their own definition and calculation formula. The design diagram of this similarity interface and its relationship with the three specific similarity tools is shown as below in Figure 2.

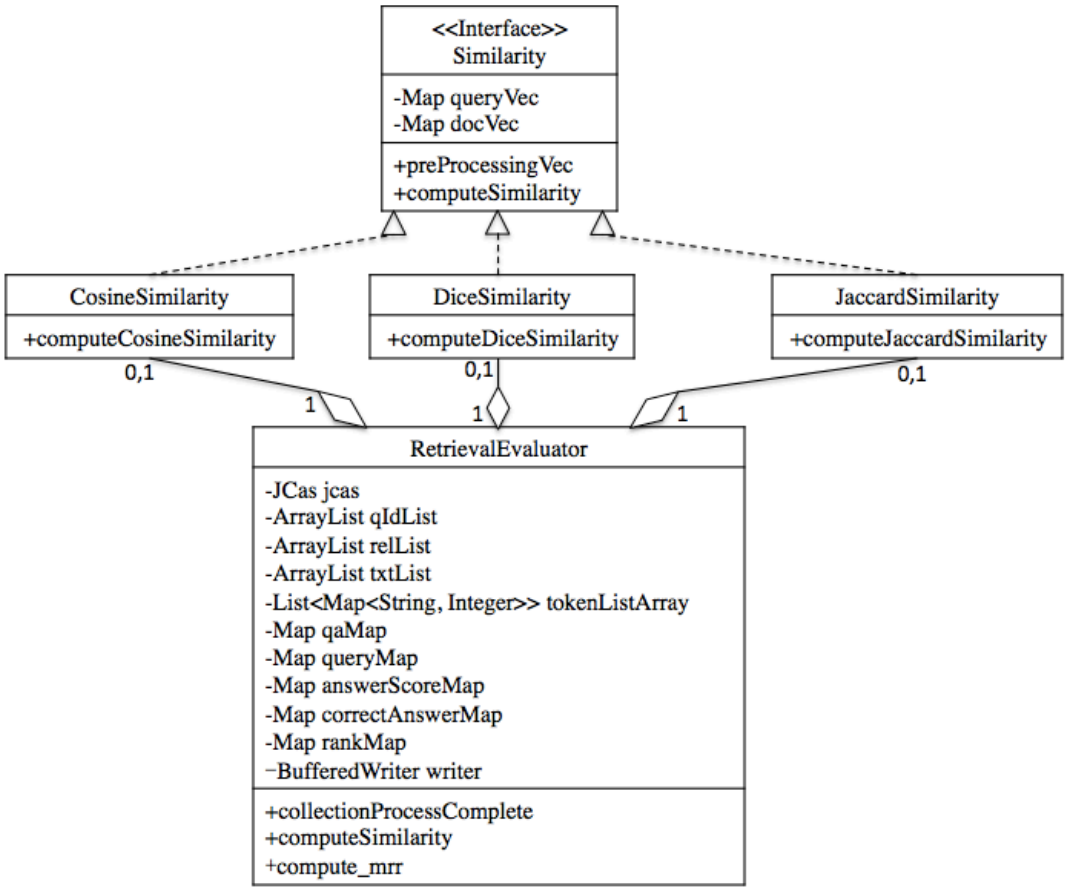

Figure 2. Similarity Computation Component Design Diagram

3.2 Dice Similarity

Dice Similarity is a statistic tool used for comparing the similarity of two samples for information retrieval, thus it's possibly an alternative for the Cosine Similarity. Its calculation formula is: $S_D = 2|A \bullet B|/(|A|^2+|B|^2) = 2p/(2p + q + d)$, where p is the number of tokens in both A and B, q is the number of tokens in A but not B, d is the number of tokens in B but not A.

Using this formula, I get the performance: MRR = 0.5012.

3.3 Jaccard Similarity

Jaccard Similarity is also a statistic tool used for comparing the similarity and diversity of sample sets, thus it's possibly an alternative for the Cosine Similarity. Its calculation formula is: $S_J = |A \cap B|/(|A \cup B|) = p/(p + q + d)$, where p is the number of tokens in both A and B, q is the number of tokens in A but not B, d is the number of tokens in B but not A.

Using this formula, I get the performance: MRR = 0.5012.

3.4 Summary

From the performance as above, we can tell that Dice and Jaccard Similarity did not working as well as the Cosine Similarity.